All right, so what we're going to do today is continue our discussion of modularity and how you hook software modules up together.

And what we did the last time was talked about how you share data between programs between users.

And we went to the design of the UNIX file system, or at least a particular aspect of the UNIX file system where we talked about how we built a file system out of layers and layers.

And each layer was essentially doing name resolution in order for us to take a UNIX path name and eventually get the data blocks corresponding to the data in that file.

What we're going to do today is to move away a little bit from the memory abstraction, which is what we spend our time on the last time, and start talking about the second abstraction -- the interpreter abstraction.

And we're going to do that in the context of software libraries, and basically understand how software libraries are put together, and how you can build large software programs out of individual software modules that get hooked together.

So that's the plan for today.

And in addition to seeing the actual mechanism for how you take all these modules in the form of libraries and other software pieces, hook them up together, what we are really going to focus on is the mechanism by which these different modules actually run on your computer.

OK, so it's going to involve a fair amount of mechanism going down to the lowest layer until you actually have something that's a single, executable file that runs on your computer.

And then what we're going to do is to step back and think about the kind of modularity that we will have gotten from this approach.

And that's the kind of modularity that we're going to call soft modularity.

So, that's the plan for today.

And you'll see why this is called soft modularity.

And it has to do with the way in which propagation of faults occurs and in particular the kind of modularity at the libraries that we are going to discuss today has the property that a problem in one module, for example, an infinite loop or a crash in one module will turn out to affect the whole program.

And the whole thing will come crumbling down.

OK, and that's why it's going to be called soft modularity.

Of course, we don't like that kind of modularity, although it is useful to have soft modularity.

So the next three lectures after today, we're going to talk about hardening the soft modularity using a variety of different techniques.

So that's the plan for the next three to four lectures.

OK, so let's take some examples to start with of where you end up using these modules -- these software modules -- to build bigger software systems.

And the first one is actually what you see in programming language.

When you use C or C++ or Java or Perl or any of these programs that you're familiar with, you end up actually building large programs out of taking lots of little programs, usually programmed in different files, and running a compiler on it, and it'll turn out to require other system software to take programs that are written in different modules and hook them all up together into a bigger program.

And this is actually going to be, we're going to learn today with examples from C or C++ programming language to find out how we build C modules together.

But, by no means are we restricted to these programming languages.

In fact, there's a lot of different examples.

Database systems provide a great example of modularity.

So if you take a system like, you might've heard of Oracle or Sybase, or any of IBM's DB2, all of these things are fairly complicated pieces of software.

But what they invariably allow you to do is to put in your own software, your own code, that will run as part of the database system.

So if you, for example, come up with, you know, normally databases allow you to run queries on tables of data.

We'll be back after these messages.

OK, so every database system allows you to upload modules into it that run as part of the database system.

So if you come up with a different kind of data type, for example, some geometric data type on which you asked whether there is a bunch of points inside a polygon, a regular database system won't actually support that.

But what you can do is write code for that, and upload it, and run it as a module in a database system.

Another example, a common example these days are Web servers, particularly something like Apache -- -- where you can actually include as part of Apache when you run it a bunch of different modules for doing a variety of new functions that weren't previously running.

And all of these different kinds of systems will turn out to use essentially variants of the same basic idea in terms of solving the problem of taking these different modules together and composing a single, large program that you can execute on your computer.

So what we're going to do today is discuss this method, this way of doing modularity in the context of, a particular example, it's easiest to do this with an example.

And then we'll step back and talk about the general principles.

So, we're going to focus on the Linux operating system running the GNU tools.

And in particular, we're going to focus on how you do this in a language like C or C++.

And so, most of you are probably familiar with some of the tools that you use to build programs in GNU and Linux and we are going to use a compiler called GCC that will turn out to use some other pieces of software to build modular programs.

So, that's kind of what we're going to start with.

And the basic approach in terms of how these modules are going to be written is if you want to write a program, you typically design what you want to do, design the system, and then decide on how you decompose the functionality of your system into a bunch of different modules.

And typically you write one or more files for each module and system, and then you need a way to bring all these modules, these different files together to build a bigger program.

And when you run GCC in particular on a dot O file, the modules themselves want to compile a C file.

As you know, you produce an object file with a name like file dot O, and that actually contains in it, that's an object file that with some more work you can arrange to run on your computer.

And what we're going to do today is figure out how when you have a large number of object files, and it will also turn out, we'll talk about something called a library, which is a collection of object files put into a single archive, or how you can take those object files and produce an executable that will run.

So if this works, let me show you an example of what we're going to be talking about.

So I have two little files, a little trivia program.

I don't know if this is visible.

OK, so all this is doing is computing the square of the number.

And it uses here, this is a very trivial program.

It defines a few variables, and then just calls on modular function called SQR, which isn't actually in this file.

It's in a separate file.

But one thing you can do is run GCC-C which just tells it to produce the object file, and there's a program called object dump which is a useful program if you want to see what's inside your object file.

It's a binary file, so you're not going to be able to hack it.

But you could run it with, we've already reached the bottom, all right.

OK, so the part to worry about for the moment are shown in the last three lines that you could see here starting main, SQR, and printf.

And those are the three functions that are being invoked by the program I showed you.

And you can see, for SQR and printf, this object file doesn't actually know where it is or where its definition is.

And that's why you see UND for undefined.

And part of what we're going to figure out today is how we can find out where those other modules are defined, and hook up all those different modules together to build a program that will actually run to do what we want it to do.

So the other thing here is square dot C, which does the obvious thing of just multiplying two numbers.

And you could do the same thing with square dot C.

So I have a little thing called show object which is just an alias for object dump.

And you can see here that there aren't actually any undefined symbols or undefined functions because square, in turn, didn't invoke anything that was outside and defined somewhere else.

And underneath here, you see the actual disassembled assembler code for this machine text.

So going back to M dot O, I just want to draw your attention to a couple of things.

Yeah?

Shows up fine on my screen, I know.

[LAUGHTER] All right, good.

OK, so there's a few sections to this object file here.

And it's important to understand a little bit how an object file is laid out.

It has in it a few different sections as I mentioned.

One of the sections is called the text section.

The text section basically contains the machine code.

I mean, ideally once you hook all these modules together, that machine code will actually just run on your computer.

It also has a section, I think you can see it up there, called auto data.

That's the second section that stands for read only data.

And one of the things I had here in that program was something that said printf with a string inside.

And, that string is read only data.

I mean, you don't want, the program doesn't actually modify it.

And that's the kind of thing you could see it on the right.

There is a comment there: the square of something is something.

And that's an example of read only data.

And then it has a section for data which really corresponds to the global variables used in the program.

And for those of you who remember 6.004, and if not we'll talk about this next time.

The look of variables in your modules are not actually in the data section.

They're typically on the stack.

So we're not going to worry about that for today.

And then there's a section called a symbol table which is shown here, I think, as sym tab not on the screen.

Let me go back.

So that's the symbol table.

And what this symbol table shows is the different global variables and functions that are either defined in the module in this dot O file or are referenced by this module.

OK, and for the symbols that are defined in this module, what the symbol table tells you is the address at which you can find it in the module.

OK, and for the things that are not defined, it says it's undefined.

And hopefully once the compiler sees all of the different object files involved, it can piece together these different object files and build a larger program module.

So when you compile each of these things in this example, M dot O and SQR dot O, you will find that the object file that's produced for both files starts with address zero.

OK, so obviously when you hook these modules up together, you can't have both modules run at address zero.

So one of the things you need to be able to do is to take these different object files together, and somehow join them so that the addresses are no longer colliding with one another.

So that's one problem that we are going to have to solve.

So that's what we're going to do today.

So there's basically three steps that we are going to talk about today.

The first step is figuring out all these different symbols.

So when you have SQR or if you were using square root in a different program, printf is another example, figuring out where the definitions of these symbols are, and where the definitions of the global variables are in your program.

And that's a step called symbol resolution.

OK, and the plan is going to be that each object file is going to have inside it a table like here that shows for the symbols that have been locally defined, where in the local module they are.

I'm going to use the word module for these dot O files.

And for symbols that are not in the same dot O file, it just says it's undefined.

And I need it.

OK, and that's what the dot O contains.

And, when you take all these dot O's together and produce a bigger program, those symbols are going to be resolved.

And that's the step called symbol resolution.

The second thing we have to do before we get a single big program is to do something called relocation.

So what relocation is, is remember I told you when you compile a program to a dot O file, all of the dot O files all have addresses starting from zero.

You can't run them all together unless you actually modify the different addresses.

And any time you see a reference to a variable in one of the dot O files, you have to modify that address to point to the actual address at which the instruction would run or the data object is present.

And that's called relocation.

And it'll turn out that the object files that are produced by the compiler are what are called relocatable object files.

Pretty much every object file these days is relocatable.

But the relocatable object file allows you to do this relocation easily because it has in it information that tells

whoever is composing these modules together how to modify the addresses that are referenced within each module.

And the third step once you do all this and you produce a single big program that's ready to execute, is actually to run the program.

And that's a step called loading or program loading.

So you create a big executable file, and you type it on the command line.

What happens?

So, those are the three steps that we are going to be talking about today.

And the first two steps typically are called linking.

OK, so most of today is going to be talking about how linking works.

And the piece of software, system software, that does linking is called a linker.

And in the GNU-Linux operating system, it's a program called LD.

That's the linking program.

And so when you run GCC to take a bunch of modules and produce a program out of it, underneath inside not visible to you but underneath, GCC actually invokes a bunch of other programs including LD as one of the last steps to produce an actual file that will run.

So the linker takes as input, object files, and produces as output a program that you can run.

And then the loader takes over and runs the program.

So there are actually three kinds of object files that the linker takes as argument.

The first kind of object file is a relocatable object file.

OK, actually there's two kinds.

The first kind is a relocatable object file.

And that's what we're going to be spending most of our time on.

The second kind of object file is something called a share object file.

We are going to get to this a little bit at the end.

And the reason we're going to wait until the end is that the reason for a shared object file is that if you have a program, a module like printf or some math library like using functions like square root, if you just do linking the normal way, which is the easy way we're going to talk about for most of today, you will end up with a program that includes in its text that includes in the binary all of the modules corresponding to all of the libraries at, you know, the text corresponds to the different modules that you use.

So now, if you have 100 different programs running on your system, all of which use printf, it will turn out that every program is quite big.

And so, shared object files allow us to not actually have to include the text of printf in every binary that we produce, but just maintain a pointer to something that says, when you need printf, go and get this, and include it as part of your program.

So we get to that in the end.

But for the most part, we are going to be focusing on relocatable object files, that is, object files that are produced by something like GCC that I showed you, which include in it information for modifying the addresses that are being referenced inside the module.

And once the linker takes these as arguments, what it produces as output, out comes an executable file.

And these dot O's don't actually run.

The executable is a thing that's capable of actually running.

OK, so the first step we have to talk about is symbol resolution.

And so the input is a set of object files.

And the output is going to be something that allows LD, the linker program, to know which symbol is located where in all the different modules.

And if your linking succeeds, then it means that every symbol that's been referenced by any of the modules that's on the command line of this LD program, there are no undefined symbols remaining.

So that's what symbol resolution does.

And the input to this is really it looks something like you have a bunch of files.

I'm going to call them F1 dot O, F2 dot O, all the way through FN dot O.

OK, and for those of you who know what libraries, don't worry about them now.

We are just going to take these individual modules, and we are going to produce out of it a single program that includes in it all of these different modules such that there are no unresolved symbols, OK?

So, SQR is one file, and M dot C contains references SQR, we want to make it so the actual program, when the reference to SQR comes in, you know where it is.

So, what each file contains when you do GCC on F1 dot CN, you get F1 dot O, it contains in it already a local symbol table.

And you saw that in the example I showed you.

And the symbol table has the following structure.

There's two kinds of data in the symbol table on a local module.

The first is a set of icons, D1, that correspond to global variables or functions that are defined in the module F1 dot C, and therefore are defined in the module F1 dot O.

So, this just tells the linker, you know, if you see the symbol being invoked by somebody else, it's being defined in this module.

OK, and it contains in it information about where in the local module it does it so that you know at what address it's being defined and so on.

Likewise, you have here another set of symbols that you want which corresponds to things that are not defined in F1 dot O.

And likewise, for each of these things you have D2 and U2, and so on, OK?

So, the way the particular implementation of the linker under GNU-Linux works, and there are many ways to implement linkers, and there are very complicated linkers, and obviously simple linkers as well.

The Linux one is particularly simple at least in terms of this way of linking, resolving all of the symbols here.

You just scan the command line from left to right and you start building up three sets, OK?

So we're going to build up three sets in this algorithm when we scan from left to right.

The first set is a set I'm going to call O.

And the idea in O is going to be all of object files that go into the program, that go into the output of the linking step.

In this case, in the end, it's going to be pretty straightforward.

It's going to be F1 dot O, union F2, all the way up to union FN.

It's going to get a little more complicated when we talk about libraries.

But for now, it's a very easy set.

When you scan from left to right, you look at the next object module or the object file, and all you have to do is O goes to O union Fi dot O.

OK, this is the I'th step of the algorithm.

OK, now we have to go to the next step, which is get to the defined symbols.

So as we go from left to right, we build up a set of defined symbols.

Initially it's null, and then we start with D1, and then we do D1 union D2, and all the way.

So up to the I'th step, what we're going to end up with is obviously at the I'th iteration, we are going to have some running set of defined symbols that have been defined in the modules so far.

And we're just going to do D goes to D union Di where Di is the set of symbols that are defined in module number I.

OK, in the last set, we're going to calculate, and the linker is going to calculate as a set U.

And U is the set of all undefined symbols so far.

So you have a set of undefined symbols.

And you are at the I'th stage of this process where up to here you have a set U of undefined symbols.

And now you're seeing this new module a bunch more undefined symbols.

So clearly what you have to do first is do union Ui, correct, because you now have built up a bigger set of undefined symbols.

But then, notice that there might be symbols that were previously undefined that are now being defined in the I'th module, right?

Otherwise, I mean, if this set kept growing, then all you would end up with is a big undefined set at the end.

So clearly there are symbols that are being defined and subsequent modules.

So you have to subtract out a set.

And the set you have to subtract obviously is the undefined set intersection whatever is being defined in this module.

OK, and so the linker does this from left to right.

It's actually a pretty simple linker because it doesn't go back and do anything.

And it doesn't actually have to at least for this way of hooking up object files together.

And what you get at the end are three sets: O, D, and U, OK?

And, the linker outputs success, it then gets to the relocation stage of U is null.

If there are no undefined symbols at the end, you know that now you can then relocate by modifying patching the different addresses together to produce your big program.

But if the set is not null, you know that no matter what you do to relocate, there is some symbol that is not being defined, which means that module won't run, which means the program won't run, OK?

So that's kind of what this linking program does.

It just goes from left to right and resolves all symbols.

And although it's done in the context of, we've discussed this in the context of a particular example of how GNU-Linux does its linking, the basic idea is the same.

Essentially, all symbol resolution will turn out to use some variant of something that greatly resembles this method.

There is actually only one problem with what I described so far, and it's wrong.

So, what's the problem?

Yeah?

Right.

If the symbol is undefined in F1 and defined in F2, you're fine because let's take this example.

I had M dots it's OK.

We're just building up sets.

So you can have a symbol that's defined.

You are saying, what if the symbol is defined in one of the files and undefined in the next file?

No, it doesn't matter for this.

It's going to matter when we talk about something called a library, but it's not going to matter here because we've built up these sets of what are defined and what are undefined?

So let's say you defined a symbol here, and you come here and you find that it references a symbol that's not locally defined, but has been previously defined, right?

That's been built up in the set, D.

So, in the end -- Oh I see, so the code is a little wrong because I have to actually update.

You're right.

I had to modify that U line.

This is wrong.

Good.

So I have to modify the set of undefined symbols to include the things that were previously defined.

So actually it should probably have been U intersection D.

And that will probably fix it.

Good.

Any other errors?

Yeah?

Right.

So, there were two.

This one I actually didn't realize.

But it's actually right.

So it should be U intersection D.

The other one is that we just keep doing D goes to D union Di.

But now, if I define a function, SQR here, and I define the same function SQR again here, we're not going to know which SQR to actually use.

So we have a duplicate symbol.

And so, in fact, what the algorithm actually ought to do is while it's doing this union computation, it better make sure that any symbol that's being defined in a subsequent module has not already been defined in a previous module.

And, if there is a duplicate definition, it'll tell you that there's a duplicate definition.

And I'm not going to show you an example, but if you just go and type out a little -- use SQR twice -- and you run the compiler, GCC, on it what you'll find is that it'll tell you that this symbol is multiply defined.

OK, so we're not going to want that either.

So once you obtain these different sets, what you'll end up with is information that will tell the linker everything about all of the different symbols, and where they have been defined, and in which object file they've been defined.

And so, the step that it has to do after that is the relocation step.

So this was the first step.

It will turn out that this step is also pretty straightforward because what happens when GCC runs on a single C file and produces a single dot O file is it contains in it information on how to relocate all of the variables and all of the functions that are defined in that module.

And there is a section of the object file that I haven't shown you called the relocation section that tells you, for any given variable in the text, or any given line of code in the text area, all of the load instructions and all of the variables, how they get remapped inside.

So that's maintained.

Basically the approach is to maintain local table.

And, it's called the relocation section of your program, OK?

So, we're going to get to loading in a little bit.

But before we get to loading, I want to get back to symbol resolution.

So here we just talked about taking a bunch of dot O files and producing a final program.

But there is another notion of something called a library.

So, how you do symbol resolution with a library?

So an example of a library here is lib C dot A, which is the standard C library.

And that's the library that contains the definition of printf.

So if you use printf or any of these other standard functions, they define the C library.

Another example is the math library, where you have the square root function and a bunch of other mathematical functions.

So we are going to want to know how to resolve with the library.

So first, we have to know what a library is.

And what a library is, is just you take a bunch of object files together and basically just concatenate them together.

It's not literally concatenation because the library also maintains an index that says, has information about what modules, what included dot O files contain which symbol definitions?

But essentially it's just a concatenation of a bunch of dot O files, OK?

And they get put together in a library.

And there is some index information in front that says what is where inside the library.

OK, so the approach to resolving symbols with a library is almost the same as what we have so far, except with one twist.

And the twist is that often libraries are extremely big, much bigger than a single dot O file.

And, one approach to resolving with libraries would be to apply the same approach here.

So when you have something that's defined in the standard C library like lib C or the math library like lib L, you could just include the entire text of the library inside to build your program.

But that just makes things extremely bloated.

I mean, think about if you just wanted to use printf in your program like we had in this example and you had to include the entire C library, which is megabytes long, it seems like not a good way of doing the linking.

So what we're going to do with the resolution of libraries is essentially the idea is to only include the dot O files that are in the library in which undefined symbols that were previously encountered are defined.

And if you think a little bit about what I said, that's one reason why it usually a good idea when you do GCC and use the linker, use LD, to specify the libraries at the end because what we're going to do is we're going to take all the dot O files, and then they're going to be things like dash LM.

I mean, the standard C library is usually, by default, already included on the command line.

But if you have functions like square root and so on, here it what we're going to do is we're going to build up.

What the linker is going to do is it is going to build up a set of undefined symbols until it gets to the end.

And there's going to be undefined symbols remaining.

If you use the square root program, the square root function, and you didn't write your own square root function, it's in the math library.

Then you might ask for the math library to be included, and you want to pull the definition of square root from The math library the way in which the linker does to pull the right dot O file is that the math library is going to have

information that says which object file contains what symbols, and anytime you see an undefined symbol at this stage, we are going to scan this archive looking for the symbol that's been undefined, in particular looking at this example for the square root symbol.

OK, and when we find the square root symbol somewhere inside in some dot O file, we're going to take that dot O file, and not the rest of the library, and then use this algorithm that we did.

So take that dot O file alone and push that as input to this algorithm.

That's the way in which we're going to build it up.

So that's why at least in this particular implementation of the linker which is very simple, if you put the LM way up in front, you are a little bit in trouble because if Fn dot O is the file that actually uses square root, and the math library was included well in front, then square root would not yet have been part of the undefined set of symbols until then.

And there are other ways to deal with it.

You can have linkers that go in more passes that presumably can deal with this problem.

But this is just an example of how GNU-Linux does this, and that's just worth knowing.

So this idea of linking and symbol resolution and relocation, people have worked on this for a very long time.

And almost every software system uses it.

In fact, if you use LaTeX to build your design papers and so on, it does symbol resolution as well because there is all sorts of cross-references that are there in LaTeX.

And it uses essentially the same kinds of algorithms, go over the files and go over the documents in multiple passes and resolve the symbol.

So it's a pretty general algorithm that we described here of building these different sets to ultimately figure out whether there are undefined references remaining at the end or not.

So I want to step back for a minute and generalize on the different approaches that we've seen so far for doing symbol resolution.

And today we saw one approach for doing symbol resolution.

But in fact, last time we saw a couple of different approaches to resolve names whose values we didn't actually

know.

So, we're going to step back and generalize with a couple of different techniques that we saw the different examples.

So the general problem here, the specifics are how you can find out where these undefined symbols are defined, or in the UNIX example, how you can take a big path name and identify which blocks contain the files or contain the data for the file that was named in the path.

But the general problem is you have a set of names.

And associated with each name is a value.

And these names get associated with or mapped to the different values.

In fact, the names get bound to the different values.

And in this example, the linker needed to take a name like the definition of a symbol and needed to identify, what's the value associated with that symbol?

The value here in this context is, where is this name, the square root function?

Where is it actually defined?

At what location is it?

OK, and so the way in which that resolution is going to be done is done and general using something called the name mapping algorithm.

And the mapping of a name to value being done by the name mapping algorithm takes into account, or takes as input something called the context.

And I'll describe this with an example in a minute.

So somebody has found a name to value.

And when you are a linker or when you are part of the UNIX file system, and you're trying to identify the value associated with the name, you're going to have to resolve that name to find a value.

And that resolution is going to be done in a particular context.

So, for example, you might have two files with the same name in two different directories, and that's fine.

The same name can have two different values because that resolution from the name to the correct value, in the directory case it's an inode number would be done in the context of the directory in which the resolution is being done.

OK, so what we've seen over the last couple of days, the last lecture, and today, are three different ways of doing it.

And it turns out that in almost every system, or in every system that I know of, anyway, there's basically three ways of doing this name resolution.

The first way -- the simplest way -- is a symbol lookup.

Look in the context of a dot O file, which has a set of defined symbols.

Taking one of the symbols, the name, in that case, as input, and finding out where it's been defined in that dot O file is basically a table lookup.

That's what that symbol table section describes.

From the disk example from last time, the inode table is an example of a table lookup.

I mean, there's a portion of disk that has in it the mapping between inode numbers and the corresponding inodes.

And that's just a table lookup.

So when you want to go from an inode number to an inode, you do a table lookup.

And that's the simplest base form, base case of how this name resolution is done.

The second way of doing name resolution is something called a pathname resolution.

We didn't see an example of this today, but we saw an example of pathname resolution the last time.

If you take a big UNIX pathname slash home slash foo slash bar, what we did was start left to right along that path and narrowed down our resolution of the file to get to the block that we wanted while going through a path, sorry, not a search path, but going to the path that names the item.

And a third way of doing name resolution is what we saw today.

And that's an example of searching through contexts.

There's really no path here.

I just tell you, here's a set of dot O files, and here's a set of libraries.

And the symbols that are undefined are defined in different modules of my program, or F.O2 in different modules of my program are defined somewhere among these modules.

And I'm not really going to tell the linker what's been defined where.

It's up to the linker to figure it out, and it does that by basically running a search running a search through a variety of different contexts.

So, each dot O file and each library is a new context in which a search for previously undefined symbol happens.

And in this particular case, the search within each context takes the form of the table lookup, OK?

So, those are the three techniques for how you do name resolution in general.

And we saw two of them today, and we saw two of them, the first two, last time when we talked about the UNIX file system.

So the last step in the process of what a linker does after symbol resolution and relocation, relocation in this particular kind of linking, I didn't use the term, but this form of linking is called static linking because we're going to take all of these different object files and defined on the command line or on the library, and build together a single big binary that has been linked once up front where the linker's called.

That's called static linking.

Internal relocation in that context is pretty straightforward.

But so we're not going to talk more about that except to note that this relocation table is maintained in each object file.

But the third step is a little more slightly more complicated, and actually varies a lot depending on the system.

And that's program loading.

And the problem that's solved by loading is that the linker produces an output program that's executable.

It's and you can run a program.

In UNIX, you run it by typing something on a command line.

Or in Windows, you click on something, and effectively that causes a shell to execute a program.

So somebody has to do the work of when you type something on the command line, somebody has to do the work of looking at what file has been typed, taking the contents of the file, loading it up into memory, passing control to something that can then start running the program.

And typically the place where that control is passed is the interpreter corresponding to the program.

So all of this work is done in UNIX by a program called execve.

So the actual loader in UNIX is a program called execve.

And its job, once you type it on the command line, is the shell invokes it.

And what it does is to do what I said, which is look at the file name, take the contents of it, load it up into memory, and pass control to basically the first line of the program.

Often, modern object files are a little more complicated.

They actually have something that says who the interpreter of the program is.

So you pass control to that interpreter, which in turn goes to a bunch of steps, and then invokes the first line in main.

And that's what the C loader, the way in which loading a program that's written in C works.

So, so far what we've seen is, as I mentioned, an example of linking called static linking where you take all these object files and libraries, and extract the right object files out of it and build a big program.

So, what you'll find is that even small programs like this one, all it's doing is multiplying two numbers.

If I compile it -- -- it's pretty big, almost 400 kB.

I mean, it's multiplying two numbers and it takes 400 kB to multiply it.

And the reason is that I made the mistake of including, I have to show the output to the user, so I call it printf.

And printf happens with part of a big object file that defines a lot of other things.

And that whole thing got built as part of the program.

Now, if you have a lot of programs, so it's not just that the files are big.

I mean, disks, as a mentioned a couple lectures ago, disks are cheap.

And so that's not really the problem as much.

The problem is that this is the entire program, so it has to get loaded in.

So if you have some machine on which a hundred processors run, and each of those processors has printf's in a few places or even one printf, each of those programs is going to be extremely big.

So the way in which you deal with this problem is to do something that's pretty obvious in retrospect.

Why have multiple copies of the same module?

Why don't we just have one copy of that module running and all the programs that use that module?

And that's done using an idea called a shared object or shared modules.

And this stuff is extremely hot.

If you look at recent activity in almost every modular software system like you look at Apache or you look at many database systems, and also you look at GNU-Linux, there's been a ton of activity over the past five or six years on different ways of optimizing things so that, the idea is a very old idea.

But there's still been a lot of activity making it efficient and practical over the last five to ten years.

The problem with shared objects is the following.

And this has become a little more clear when we talk about actually something called an address space a couple lectures from now.

But the basic problem is that instructions are associated with memory locations, and they run each thing in the object file.

And the binary has an instruction location.

And data has a particular location associated with it as well.

And the problem is that when you have two programs that each want to use a shared module, unless you're really careful about how you design it, it's going to be very hard for you to ensure that for both of those programs, this module that's going to be shared has exactly the same addresses because if you have in one program the module being called from address 17, and in another program the module is written up as 255, then that object cannot be shared, right?

It's two different objects.

And that's what happens with static linking.

If you take the SQR.O module and you include that in two different programs, the addresses that get associated with it in the two different programs are going to be completely different.

So one challenge, and a significant one in the shared object is objects that are shared by different programs running at the same time is to generate code that is what is called position independent.

So it doesn't have in it anything that's different for the different programs.

And so this is called position independent code.

So the idea is when you call the module on your computer, the program counter is going to point to something.

And all of the addresses inside that module are going to have addresses.

Obviously, they're going to have addresses, but they're going to be relative to the program counter.

So when you jump to a location, I'm going to jump to 317. You're going to jump to something that says five locations from where you are now.

So it's a kind of addressing called PC relative addressing.

And that's not going to be the only thing that's used.

But by and large, a requirement for position independent code is that all of the addressing be relative to, say, the program counter.

And once you have this kind of position independent code, what you have to do in your program, if you're using something like the square root program, let's say, OK, in the math library when you do the linking, you don't have to include the object file in which square root is defined.

All that your object file has to do now is at the time it was linked, there is some library, runtime library, that you

know when you link the program has the definition of square root.

So what the program contains when you run this F1 dot O, F2 dot O, all the way through the library is it's just going to maintain a pointer, a name actually.

It's going to maintain a name to where the square root function is defined.

It's going to be a filename.

OK, and so this is why sometimes when you type in a program, and somebody has changed the configuration on your machine and built before, and somebody changes the configuration, sometimes you get an error message that while you're running the program, you get an error message that says some library dot SO not found.

It worked two days ago.

It doesn't work now.

And the reason for that is somebody may have moved things around and you get an error not when you compile the program, but when you run the program.

And many executions of program may not actually trigger the error at all.

It may get triggered only when the actual object is needed.

And that's called dynamic linking.

And again, the way in which we do dynamic linking is to use the same name and constants.

Rather than embed the entire object file corresponding to the module corresponding to a name of a function that's being defined elsewhere, maintain a reference to it.

And load that reference up at runtime.

Now, in order to enable for that object to be shared between different programs, all of the addressing inside that has to be relative.

That is, it has to be independent of what the PC's value for the starting point of that module is.

And that's called position independent code.

So that's the basic story behind how linking works and almost all software systems involving libraries and modules

ends up having a linking in it.

I mentioned LaTeX as an example before.

Even document systems have linking in it.

And it's a pretty fundamental algorithm that we talked about.

It's specific to GNU-Linux, but the basic idea is pretty common.

Now, what we'll see next time is that this way of modularizing has a lot of nice properties, allows you to build big programs, but at the same time has pretty bad fault isolation properties that we'll talk about next time.